Scripting a Paint-like Application Walkthrough

One of the key features Roslyn adds for C# Interactive is the ability to execute snippets of C# code in a host-created execution environment. This walkthrough shows a C# application that uses the Roslyn Scripting APIs to let the application’s users write code to operate on the host's object model, provide callbacks, and bind variables to host UI elements. The application is a very simple paint application written in WPF.

This walkthrough shows you how to:

* Use the Roslyn Scripting APIs to create a C# script engine. Engines represent a language implementation (C# or VB) that can execute code on behalf of a host application.
* Create a Session, which holds the cumulative execution context for script code across several separate executions of code snippets.
* Execute C# code in a host-supplied scope.
* Push top-level function definitions as callbacks or command implementations.
* Add default references and namespace 'using's for C# script code allowing script code to access the host’s extensibility DLL and namespaces.

**Highlights of the code …**

Open the PaintLike project in Visual Studio, which installed to your documents folder under "Microsoft Roslyn CTP - September 2012\CSharp\PaintLikeScripting\PaintLikeScripting.csproj". This project defines the PaintLike UI in xaml. The UI has an input pane for submitting code snippets to execute against the host's object model. Note, this walkthrough will not explain every detail of the PaintLike application since most of it is WPF UI code and timer/thread management to let a callback function create animations by moving objects.

Looking at MainWindow.xaml.cs (which is under MainWindow.xaml in the Solution Explorer), you can see the code has 'using' directive to be able to access Roslyn Scripting APIs:

using Roslyn.Compilers;

using Roslyn.Scripting.CSharp;

using Roslyn.Scripting;

The PaintLike application defines an object model, which enables script code to manipulate and interact with the application as a host. The PaintLike application puts an instance of the HostObjectModel type in a Session (shown below). The Roslyn Scripting APIs for executing code bind free identifiers in script code to public members on the host object.

There have to be public members on the application’s host object to get at private or internal code of the host. For example, the HostObjectModel.Selection property is public so that the script code can access the application window’s Selection property. The host could decide to just expose the main window object rather than selectively exposing individual pieces of functionality. The PaintLike application also exposes Application, just as an example of OLE Automation style where the host’s primary objects have a self-property named Application.

public class HostObjectModel

{

private readonly MainWindow window;

public HostObjectModel(MainWindow window)

{

this.window = window;

}

public HostObjectModel Application { get { return this; } }

public Canvas Painting { get { return this.window.canvas; } }

public Shape Selection { get { return this.window.selected; } }

public Action Callback

{

get { return this.window.callback; }

set { this.window.callback = value; }

}

public UIElement PaintingChildAt(int i)

{

return this.window.canvas.Children[i];

}

} // HostObjectModel Class

The MainWindow constructor creates a C# ScriptEngine to execute user code snippets. ScriptEngines represent the language and execution semantics, and they hold onto some general context such as references for compilation or seeding new Sessions. The sample code fragment below supplies several references that the script code will need to execute. Rather than force end users to provide #r code snippets in their scripts, the host establishes these references for the code. The references passed to the ScriptEngine include a reference to the host's assembly so that script code can bind to the host object's public members.

// Initialize Roslyn hosting with references and using.

csharpEngine = new ScriptEngine();

csharpEngine.AddReference("System");

csharpEngine.AddReference("System.Xaml");

csharpEngine.AddReference("PresentationFramework");

csharpEngine.AddReference("PresentationCore");

csharpEngine.AddReference("WindowsBase");

csharpEngine.AddReference(this.GetType().Assembly.Location);

session = csharpEngine.CreateSession(new HostObjectModel(this));

The PaintLike application creates a Session to hold the cumulative effects of executing script snippets. Sessions represent cumulative execution context for a group of definitions (variables, functions, types, etc.) that can all work together, but you can add the definitions to the Session incrementally or all at once. A Session also supports redefining variables, functions, types, etc., by simply executing new definitions. If you redefine a name, executing new code snippets that reference the name bind to the new definition. A Session can also hold onto an instance of the host object model so that free identifiers in hosted script code can bind to accessible members on the host object to interact with the host.

The PaintLike application supplies an instance of the HostObjectModel so that free identifiers in the script code can bind to public members on the host object.

session = csharpEngine.Create(new HostObjectModel(this));

The application UI has a button to run user code snippets, and its RunClick event handler has a single line in it:

csharpEngine.Execute(codeTextBox.Text);

You can name shapes on the host's drawing canvas so that script code can operate on the name and save references to shapes. The demo code is a bit of a hack for demonstration purposes in that it creates a variable declaration statement and executes that in the Session. If you do not want to understand the gritty details, glance at the code below and move on. The PaintLike application binds a variable on every TextChanged event from the edit box. The event handler also sets to null the previous name to avoid creating N pointers to an object when the name has N characters in it. The TextChanged handler looks like this:

private void SelectedItemName\_TextChanged(...) {

...

string oldName = selected.Name;

selected.Name = SelectedItemName.Text;

session.Execute("using System.Windows.Shapes;");

if (oldName != "")

{

session.Execute(string.Format("Shape {0} = null;", oldName));

}

if (selected.Name != "")

{

session.Execute(string.Format("Shape {0} = Selection;",

selected.Name));

}

...

The last scripting feature of the PaintLike application is the ability to assign a callback function that PaintLike will run on a background thread for animation effects. The walkthrough demonstrates its usage below.

**Walkthrough of Scripting Snippets for PaintLike …**

The steps of the walkthrough below are a demo of scripting the PaintLike application to gain a feel for the power of what script code could do with a given host object. The code snippets below are not directly relevant to the Roslyn Scripting API, nor do they try to teach you anything about WPF. However, they do show how powerful hosting C# as a scripting language and letting users define code to manipulate a .NET application can be.

**1.** Launch the PaintLike application with F5.

**2.** Click and drag a shape from the left pane onto the middle black canvas.

**3.** Click the shape to ensure it is selected (should show with a faint blue outline).

**4.** Copy and paste the following code (which you can also find in the demo\_snippets.csx file) into the bottom pane of the PaintLike application:

using System.Windows.Media;

Selection.Fill = Brushes.Red

Click Run Code, and you see the selected shape turn red. The identifier "Selection" binds to the HostObjectModel instance, which in turn fetches the selection from the WPF window.

**5.** Now enter a name in the "Selected Item" text box, such as "foo" without the double quotes. Then double click "Selection" in the pasted code snippet and change it to "foo". Replace "Red" with "Chartreuse" or some other color. Lastly, you can optionally delete the 'using' line because the Session keeps cumulative execution context, and the 'using' is already in effect. Press the Run Code button, and the shape turns bright green.

**6.** To see more you can do with the host application, paste the following code (from demo\_snippets.csx). Use ctrl-a to select all of the code that is in the input pane to replace the old code with the following:

using System;

using Canvas=System.Windows.Controls.Canvas;

using System.Windows.Shapes;

Random rand = new Random();

for (int i = 0; i < 100; i++) {

Rectangle rect = new Rectangle();

rect.Width = 20;

rect.Height = 20;

rect.Fill = Brushes.Blue;

Application.Painting.Children.Add(rect);

Canvas.SetLeft(rect, rand.Next((int)Application.Painting.ActualWidth));

Canvas.SetTop(rect, rand.Next((int)Application.Painting.ActualHeight));

}

Click Run Code, and the above snippet draws 100 blue rectangle shapes on the canvas. In this code, the identifier "Application" binds to the member on the HostObjectModel instance, which in turn just returns itself. This is an example of classic OLE Automation style object models where the host object has a self property called Application. You could leave off "Application", and "Painting" would bind to the property of that name on the host object.

You may also notice that you do not need "using System.Windows.Media" in this step. This is because the execution context is cumulative, and you already entered a 'using' for this namespace in the previous submission.

**7.** Before we begin the next segment of the walkthrough, click the Clear button to remove all the objects from the canvas.

**8.** Use ctrl-a to select all the code that is in the input pane already so that when you paste the new code below, it will replace the old code. Copy and paste the following code from demo\_snippets.csx:

using System.Reflection;

PropertyInfo[] brushProperties = typeof(Brushes).GetProperties();

int dim = (int)Math.Min(Painting.ActualWidth - 20,

Painting.ActualHeight - 20) / 2;

for (int i; i < 36; i++) {

int angle = i \* 10;

Rectangle rect = new Rectangle();

rect.Width = 20;

rect.Height = 20;

rect.Fill = (SolidColorBrush)brushProperties[i].GetGetMethod().

Invoke(null, null);

Painting.Children.Add(rect);

Canvas.SetTop(rect, dim \* Math.Sin(angle \* Math.PI \* 2 / 360) + dim);

Canvas.SetLeft(rect, dim \* Math.Cos(angle \* Math.PI \* 2 / 360) + dim);

}

Click Run Code, and the above snippet of code draws a circle of multi-colored squares. It uses a couple of bits of odd code. The first quirk, at a high level, is that the script code uses reflection to map over the colors available on the Brushes type. There's no simpler way to iterate those colors.

Notice this code snippet uses "Painting" directly to bind to the host object's property, rather than using "Application.Painting" as the previous snippet did.

**9.** For the last piece of the walkthrough, use ctrl-a to select all the code in the bottom input pane of the PaintLike application. Then copy and paste the following code (which is also in demo\_snippets.csx) into the PaintLike application’s input pane and click Run Code.

using UIElement=System.Windows.UIElement;

void callback () {

foreach (UIElement child in Painting.Children) {

// Note, cannot use Application.Painting here if 'using'

// all symbols from System.Windows. The 'using' shadows the

// host object member name.

double run = (Canvas.GetLeft(child) - dim) / dim;

double rise = (Canvas.GetTop(child) - dim) / dim;

double angle = Math.Atan2(rise, run);

angle = angle + Math.PI / 100;

Canvas.SetTop(child, dim \* Math.Sin(angle) + dim);

Canvas.SetLeft(child, dim \* Math.Cos(angle)+ dim);

};

}

// Callback is host object property for function to run.

Callback = callback;

Click Run Code, and the above snippet makes the circle of square rotate. You can click the Pause button to stop the rotating circle, and click Resume (name changes when paused) to continue. The PaintLike program is continually calling the callback delegate whenever it is not null. The identifier 'Callback' binds to the property on the host object PaintLike created the Session object with before executing any user snippets against the Session.

That's it. Please play around with the Roslyn Scripting APIs and provide feedback. See the [Roslyn Scripting API document](http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=294856) for features we're planning but have not yet implemented, but there is a lot more work to do on the scripting APIs that is not yet reflected in the document.